ELP, projet Haskell

Tristan Roussillon

26 novembre 2019

Le but de ce projet est de développer en Haskell un outil qui transforme un programme écrit dans un langage inspiré des langages Logo et Haskell en un programme écrit en SVG.

Il doit être réalisé en binôme et à rendre sous la forme d'une archive au format `zip` contenant un fichier haskell commenté (explications nécessaires à la bonne compréhension du code, à la compilation et à l'utilisation correcte de l'exécutable). Le nom de l'archive doit être composé du nom et prénom des deux étudiants et être déposé sur Moodle au plus tard le 29 Novembre 2019, 23h55 pour les 3TC, le 6 Décembre 2019, 23h55 pour les 3TCA.

1 Le langage LOGOSKELL

Le langage LOGOSKELL (version 1.0) sert à exprimer le chemin que va suivre un crayon pour dessiner un graphique. Il comporte 4 instructions, toutes avec paramètres : `Forward`, `Left`, `Right` et `Repeat`. L'instruction `Forward x` fait avancer le crayon de x points dans la direction courante. L'instruction `Right x` (respectivement `Left x`) fait tourner la direction courante de x degrés à droite (respectivement à gauche), dans le sens des aiguilles d'une montre (respectivement dans le sens inverse). L'instruction `Repeat x [ yyy ]` répète x fois la suite d'instructions entre crochets. Les instructions sont séparées par des virgules et le programme entier se trouve entre crochets et sur une seule ligne de texte.

Le programme suivant décrit la Figure 1 :

```
[Forward 100, Repeat  4 [Forward 50, Left 90], Forward 100]
```

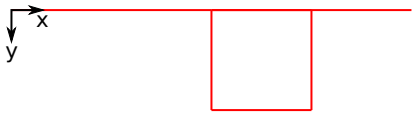

FIGURE 1 – La représentation graphique décrite par le programme LOGOSKELL donné, avec comme point et direction de départ l'origine et l'angle nul.

La même FIGURE 1 est décrite par le programme SVG suivant, interprétable dans n'importe quel navigateur :

```
<?xml version="1.0" encoding="utf-8"?>
<svg xmlns="http://www.w3.org/2000/svg" version="1.1" width="200" height="200">
<title>Exemple</title>
<line x1="100.000" y1="100.000" x2="200.000" y2="100.000" stroke="red" />
<line x1="200.000" y1="100.000" x2="250.000" y2="100.000" stroke="red" />
<line x1="250.000" y1="100.000" x2="250.000" y2="150.000" stroke="red" />
<line x1="250.000" y1="150.000" x2="200.000" y2="150.000" stroke="red" />
<line x1="200.000" y1="150.000" x2="200.000" y2="100.000" stroke="red" />
<line x1="200.000" y1="100.000" x2="300.000" y2="100.000" stroke="red" />
</svg>
```

Votre objectif est d'écrire en Haskell un programme capable de lire en entrée un programme LOGOSKELL, comme celui donné plus haut, et d'écrire en sortie un programme SVG représentant la même figure.

2 Méthodologie

Le projet peut être décomposé en deux étapes :

— Développement d'une structure de données représentant un programme LOGOSKELL,

— Traitement de cette représentation intermédiaire pour produire un programme SVG.

2.1 Étape 1 : Représentation intermédiaire

Vous devez proposer une structure de données pour stocker votre programme LOGOSKELL en mémoire, puis pouvoir produire du code à partir de cette représentation intermédiaire. Pour commencer, vérifiez que :

— Un programme LOGOSKELL est une liste d'instructions.

— Chaque instruction possède un ou plusieurs paramètres.

— Ces paramètres peuvent être des entiers ou des programmes (voir `Repeat`).

Le programme LOGOSKELL précédent peut être considéré comme un programme haskell valide à partir du moment où il existe des constructeurs de valeur correspondant aux 4 instructions `Forward`, `Left`, `Right` et `Repeat`. Cela fait partie de votre travail de définir les constructeurs et types correspondants. Par exemple, vous pouvez définir le type `Instruction` ainsi :

```
data Instruction = Forward Int
```

et alors traiter dans un programme haskell n'importe quelles valeurs du type `Instruction` comme `Forward 100` ou d'un type paramétré par `Instruction`, comme `[Forward 50, Forward 50]` qui est de type `[Instruction]`. Nous ne sommes déjà pas très loin des spécifications du LOGOSKELL [1].

De plus, nous ne voulons pas écrire un programme LOGOSKELL dans le code du compilateur, mais plutôt lire ce programme à l'exécution. Dans ce cas, il est représenté dans le code par une chaîne de caractère. A l'échelle d'une instruction, la question qui se pose est comment passer d'une chaîne de caractère représentant l'instruction en une valeur de type `Instruction` ? ou concrètement, comment passer de la chaîne de caractère `"Forward 100"` à la valeur `Forward 100` (attention à la présence ou absence des guillemets) ? Il existe une fonction, appelée `read`, qui fait cette transformation si le type en question (`Instruction` dans cet exemple) appartient à la classe `Read`, ce qu'on indique en ajoutant la clause `deriving` à la définition du type :

```
data Instruction = Forward Int deriving (Show, Read)
uneInstruction = (read "Forward 100" :: Instruction)
```

Enfin, il est nécessaire pour la seconde étape de pouvoir extraire les paramètres d'une instruction. Autrement dit, comment, étant donnée une variable de type `Instruction`, accéder à l'entier caractérisant l'instruction ? Simplement par *pattern matching* dans une équation ou une *case-expression*. Par exemple :

```
obtenirDescription :: Instruction -> String
obtenirDescription (Forward x) =
  "Deplace le crayon de " ++ (show x) ++ " unites"
```

1. Un constructeur appelé `Left` ou `Right` produira une erreur car il existe déjà un type (`Either`) défini à l'aide de constructeurs nommés `Left` et `Right` automatiquement importés du Prélude. Pour lever le conflit de nom, il suffit de cacher les constructeurs issus du Prélude à l'aide de la ligne de code `import Prelude hiding (Left, Right)`, placée en première position.

ce qui est équivalent à

```
obtenirDescription :: Instruction -> String
obtenirDescription inst = case inst of
  (Forward x) -> "Deplace le crayon de " ++ (show x) ++ " unites"
```

3 Étape 2 : génération d'un fichier SVG

Vous devez écrire une fonction qui prend en entrée une représentation intermédiaire, ainsi qu'un crayon caractérisé par une position et une direction, et retourne un programme SVG. Il y a de nombreuses possibilités et vous êtes libres de choisir celle qui vous semble pertinente.

L'une d'elle consiste à séparer la tâche de traduction des tâches d'entrée-sortie et d'écrire une fonction pure qui utilise un accumulateur pour stocker les lignes du programme SVG calculé.

```
logoskell2svg :: Programme -> Crayon -> [String]
  -> (Crayon, [String])
```

où je suppose que les types `Programme` et `Crayon` ont été correctement définis. Remarquez que le crayon et la liste des lignes SVG déjà connues est un contexte de calcul dont on voudrait habituellement modifier l'état au cours de l'exécution. Comme Haskell est un langage pur n'autorisant pas de mise à jour, la stratégie courante consiste à passer en paramètre et retourner le contexte de calcul.

Il reste alors à écrire une seconde fonction qui affiche la liste des lignes SVG sur la sortie standard [2].

```
outputStrLst :: [String] -> IO()
```

Plutôt que stocker les lignes du programme SVG calculé, vous pouvez aussi ne stocker que les extrémités des segments de droite. Dans ce cas, la signature des fonctions ressemblera à :

```
logoskell2svg :: Programme -> Crayon -> [Point]
  -> (Crayon, [Point])
outputPtLst :: [Point] -> IO()
```

Tout ce que vous lisez dans cette section ne sont que des indications et votre solution peut tout à fait être différente.

2. Si vous ne savez pas ce que sont l'entrée et la sortie standards, les redirections de flux, consultez la page suivante : `https://www.tuteurs.ens.fr/unix/shell/entreesortie.html`

A Exemple

J'ai écris un tel programme en moins de 60 lignes. L'éxécutable appelé `compilateurLogoskell` lit un programme LOGOSKELL sur l'entrée standard et écrit le programme SVG équivalent sur la sortie standard :

```
./compilateurLogoskell < prog.logo > prog.svg
```

où le fichier `prog.logo` contient par exemple :

```
[ Repeat 36 [ Right 10, Repeat 8 [ Forward 25, Left 45 ] ] ]
```

Et voici comment s'affiche dans ce cas le fichier `prog.svg` obtenu :